Fabrice Normandin, 260636800
Marcel Morin, 260605670

Group 20
Due: February 19, 2018

ECSE 426 Lab 2 Report

1. Abstract

The goal of this lab was to successfully generate and acquire data using our STM32F407VG board's build in components, along with the implementation of data filtering and computation to produce information to be displayed to a user. With these challenges in mind we were tasked with implementing an efficient way to gather data sampled by our Analog to Digital Converter (ADC) without overloading our CPU all while filtering the data as it comes in. We came to realize that although an interrupt based approach could significantly reduce CPU usage, an even more efficient approach would be to use the DMA setting of our ADC, to have it place data at a predetermined location in memory. We later filtered and ran our required computations on that data. Knowledge of the Systick interrupt timer mechanics and of the ADC callback functions were crucial for the success of this lab and for an efficient solution with respect to CPU usage.

2. Problem Statement

Our gathered data required for this lab needed to firstly be generated in a controlled manner in order to be able to later test the effectiveness our filter and data calculations. This required us to generate an analog signal using the on board Digital to Analog Converter (DAC) and have our ADC sample this signal at 50Hz. Not only were we required to sample the signal 50 times a second, but we needed to do so in an interrupt based manner that required us to not constantly be polling our ADC for values.

With the collected ADC values our next challenge was to filter our data with an Finite Impulse Response filter to obtain a smoother data set and reduce noise. With this smoother data obtained we then needed to compute the minimum, maximum and Root Mean Squared values. The min and max values needed to be constantly updating over a window of the last 10 seconds of data acquired and the RMS needed to be the most updated value (we interpreted it as the RMS of the last second of data acquired). This was a challenge as all the while our ADC's values needed to be collected while we filtered, and computed on past values acquired simultaneously.

The final problem to implement was that our computed values needed to be translated onto a 7-segment LED display that would be able to switch between the minimum, maximum, and RMS value via the push of a button on our board. These values would also need to be updated and shown as a computed voltage rather than the digital value produced by the ADC.

3. Theory and Hypothesis

As previously mentioned, one of the major tasks of this lab was to successfully acquire analog data at a rate of 50Hz. We opted to use the ADC in a 12-bit resolution mode, although in theory, the values we were to display would only need 2 decimal digits of precision, hence we could have opted for a more efficient 8-bit resolution for the ADC. Since 3.0/(2^8) is about 0.01, a difference of one binary digit would register on the display properly, had we chosen 8 bits of resolution. However, with 12 bits, a change of one binary digit only represents 3.0/(2^12), or about 0.0007V, which does not affect the displayed value. Hence, the increase in resolution is not entirely justified. This decrease in precision would have enabled a higher sampling frequency, since the Successive-Approximation-Register (SAR) ADC would only need to produce 8 values instead of 12. In our case however, the ADC's sampling time was set to 460 clock cycles, making one full value reading about 5520 clock cycles long, which at our clock frequency of 186MHz, gives a maximum theoretical sampling frequency of about 33.7 KHz. Given our very low sampling frequency requirement of 50Hz, having such a slight increase in the sampling time was deemed inconsequential, and we proceeded onwards with other aspects of the Lab.

4. Implementation

The implementation of this labs was broken into several stages that we approached in order and build off of each other as we advanced throughout. These stages consisted of: DAC initialization, Blue button interrupt, ADC interrupt implementation, Systick interrupt ADC sampling frequency, ADC DMA configuration, data filtering with FIR, calculation of min/max/rms over sliding-window, and finally 7-segment LED display implementation.

Our process started with the implementation of the DAC which needed to generate a analog signal. Looking at Figure 2 in the appendix we can see that the DAC was configured to pin PA4 in CubeMX. This didn't interfere with other crucial components of our design and was closely located near our ADC input pin as to easily allow us to connect the two together. No further initialization was required for out DAC other than having it initialized by default with the generated CubeMX code in Keil. Starting the DAC was then a matter of Setting its value using HAL_DAC_SetValue and starting it using HAL_DAC_Start. SetValue was given the ADC handler as a parameter along with the channel that our DAC was outputting to, the data alignment, and the set 12 bit value to convert to analog.

In order to better understand how interrupts worked on our microprocessor we set out to implement the blue button interrupt handler so that when pressed an interrupt flag would be set letting our program know that the button was pressed without continuously polling its pin value to see if it had been set to high. We firstly needed to assign the GPIO_EXTI0 interrupt on the blue button pin PA0 as seen in Figure 2 in the appendix. With the pin assigned we then needed to enable its interrupt under the Nested Vectored Interrupt Controller (NVIC) settings in CubeMX. In Keil, the PA0 pin would have its associated EXTI0 handler interrupt priority set and enabled letting us use the called function EXTI0_IRQHandler() in the STM32f4xx_it.c file. At first we used this interrupt handler to change the four on-board LEDs to confirm our

implementation was correct, and only later into the lab did we change its function to change the displayed values of our computed data.

Our next stage involved initializing our ADC in interrupt mode. The settings initialized in CubeMX can be seen below in Figure 1. Our bits and data alignment needed to be specified. Scan continuous conversion mode was disabled as we were only working with one channel to convert our ADC data and did not need to convert multiple analog channels with the same ADC. Continuous conversion mode was disabled to enable us to control the sampling frequency of the ADC. We were also not using DMA at this stage in our experiment. Finally, *End of Conversion Selection* was set to "EOC flag at the end of all conversions", meaning that we could convert multiple values after one another without having to stop and restart our ADC. Before generating our ADC code for Keil we needed to make sure, like with the blue button interrupt, we enabled the NVIC setting for adc1 global interrupt enable. With these settings finalized in CubeMX we could implement how our ADC collected data in Keil.

ADC_Settings	
Clock Prescaler	PCLK2 divided by 4
Resolution	12 bits (15 ADC Clock cycles)
Data Alignment	Right alignment
Scan Conversion Mode	Disabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
End Of Conversion Selection	EOC flag at the end of all conversions

Figure 1 - ADC CubeMX settings

Gathering data from the interrupt routine of our ADC required the help of the System Clock timer (systick) interrupt and the implementation of a weakly defined function ("*HAL_ADC_ConvCpltCallback()*") in the HAL_ADC library. The systick interrupt was by default called every 6 microseconds. This was determined via experimentation in the Keil debug mode using timers to time each iteration of the systick interrupt and through the CubeMX clock configuration, Figure 3 in appendix, which shows the Cortex system timer to be divided by "/1". Changing the denominator to "/8" would cause the following clock config line in our generated code to change from HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000), divided by 1000, to divided by 8000. The HAL_SYSTICK_Config function sets the period of the systick interrupts per second and 168Mhz/1000 gives you 168000 period per interrupt, or 1ms per systick. The simple fix to get the ADC to sample properly would be to alter the denominator of the HAL_SYSTICK_Config function to get the systick interrupt to occur at a rate of 50Hz. This, however, isn't in a section of code that enables the user to write without having CubeMX regenerate over it were we to re-configure settings in CubeMX, and, were there to be other peripherals that later in the experiment were depended on the systick, the interrupt going at such a slow limiting rate would not be beneficial to our system. To solve this we kept the default systick configuration interrupt as initialized by CubeMX, and instead implemented a counter

within the systick interrupt to make sure that the ADC_Start_IT function was called at a rate of 50Hz, by counting up to 20 systick calls before firing the HAL_ADC_Start_IT() function and resetting the counter value to 0.

With the ADC now able to be called at the proper sampling rate, we could finally store our data into a buffer at every interrupt. To do this required us to define the function HAL_ADC_ConvCpltCallback in our main.c file. This function was called every time the ADC interrupt was triggered and ready to give out a converted value. The ADC_GetValue function would then be called within it and added to an ADCBuffer array that stored the gather data at every interrupt while incrementing its value after every ConvCpltCallback to fill up the array. We decided to store up to a second of data (50 samples) before looping back (mod 50) and overwriting the old entries at the start of the array. We were now able to continuously store data sampled from our ADC and to check whether the sampled data entered into our ADCBuffer was correct we connected the DAC pin PA4 to the ADC pin PA1 and could see through the debug window that our retrieved values were the same +- 5 of what our 12 bit DAC_SetValue was.

Before continuing with the data calculation, we decided to implement DMA instead of the ADC interrupt as a means to sample our values. To do this required us to go back into CubeMX and enable DMA requests from our ADC peripheral. Doing this would generate a DMA global interrupt in Keil that we realized would trigger when the DMA buffer was halfway and completely full when the HalfConvCpltCallback and ConvCpltCallback functions would be called respectively. In doing this we did not need to have the ADC_GetValue inside the ConvCpltCallback function anymore as DMA was taking care of that. Instead once the ConvCpltCallback was called we just for looped through the full ADCBuffer and applied our FIR filter function per loop iteration and respectively added the filtered value to the filtered_ADCBuffer array. Making sure DMA was in circular mode would mean that DMA would loop back around to the start of the memory we allocated it. Replacing HAL_ADC_Start_IT in our main right before the while loop with HAL_ADC_Start_DMA and passing through the pointer to our ADCBuffer and the buffer size is all that is needed to start the DMA process. The systick interrupt handler still controls the rate at which we are sampling our data and filling up the buffer so no changes were required there. However, an issue arose after the first second of data had been acquired, as the DMA seemed to have stopped. Although we considered looking into using the *DMA_Continuous_Request* setting, we, in the interest of time, instead used ADC_Stop_DMA() followed by ADC_Start_DMA() once every second, which would successfully gather data for the next second with DMA.

The next stage of our experiment was to filter the incoming data using our previous lab's FIR filter. This filter takes the received ADC sampled value and applies a filter over the previous 4 gathered samples by adding all 5 values with a weight of 0.2 on each value. We implemented this using a buffer in a circular queue and added the newest sampled element at the head pointer (making sure to mod5 back around to overwrite the 6th oldest value with the newest sampled value). We then added up all the elements going from the head to the tail and returned

that as the filtered value of our FIR function. This value was then added in our ADCBuffer instead of the original value the ADC_GetValue function returned.

Then later came the task of storing the Max and Min values of the past 10 seconds, along with the latest RMS value, which we assumed to be that of the last second. We implemented this by first computing the min, max and RMS of the last second's data using the asm_math assembly code from Lab 1, then stored the results in an array, using head and tail pointers in order to create a circular queue. This array, with a capacity of 10 for max and min values (one array per value) had a new set of Max and Min inserted every second. The RMS value was also stored and overwritten every second. During each call of our *adc_buffer_full_callback()* function, the min and max of the last 10 seconds were also computed, and a global static integer variable (*display_mode*) was used in order to set *displayed_value*, a global static float variable, to the required value (either min, max or RMS), depending on the current display mode, which was changed using the blue button, and indicated with a set of three LEDs, which would alternate depending on which mode was currently active.

Then, once the *display_mode* and *displayed_value* were set, a simple *refresh_display()* method was implemented, which gets called by the SysTick interrupt handler at a customizable frequency (generally about 200Hz). This method was responsible for converting the float *displayed_value* to a set of three digits, and then alternate in showing one of these three digits on the 7-segment display. To achieve this efficiently without using delays, our simple approach consisted of calculating the segments required for each digit, and then turning on one of the digits once out of three calls to the function, alternating from digits 0, 1 and 2. A set of very helpful macros were implemented within the "*heads_up_display.h"* file, which greatly facilitated setting the GPIO pins. The Output Data Register (ODR) was used to set multiple GPIO pins at once, and also helped in having the program be efficient by bypassing the hardware abstraction layer.

5. Testing & Observations

In order to test the conversion accuracy of our ADC, we used an oscilloscope to view the voltage that our DAC was outputting, as well as that of the 3V VDD pins on the board. At first, we were pleased to see the values approximately line up, with the ADC value chosen to be about half the range (2048 out of 4095), we were seeing about 1.6 volts, when the oscilloscope was showing around 1.53V. We were then thankfully informed that our assumption of 3.3V for the maximum DAC voltage was wrong, as the VDD lines were at 3V. By simply fixing the coefficient in our *DigitalToAnalogValue* function, we then got the value we displayed to correspond precisely to that shown by the oscilloscope.

6. Conclusion

Through the efficient use of interrupts and DMA we were able to sample our DAC's set value via our ADC and then compute the minimum, maximum, and rms values of the past 10

seconds. We made sure to have each stage properly working, as described in our implementation section, before moving on to the next step, which made it easy to debug our code to find the source of errors. This experiment shed light on the importance of interrupts and lightening the CPU load for repetitive tasks that could be done by peripheral controllers. This paves the way for future labs giving the proper understanding of the ADC, DAC, GPIO pins and interrupt handlers that will be useful to implement in future experiments.

APPENDIX

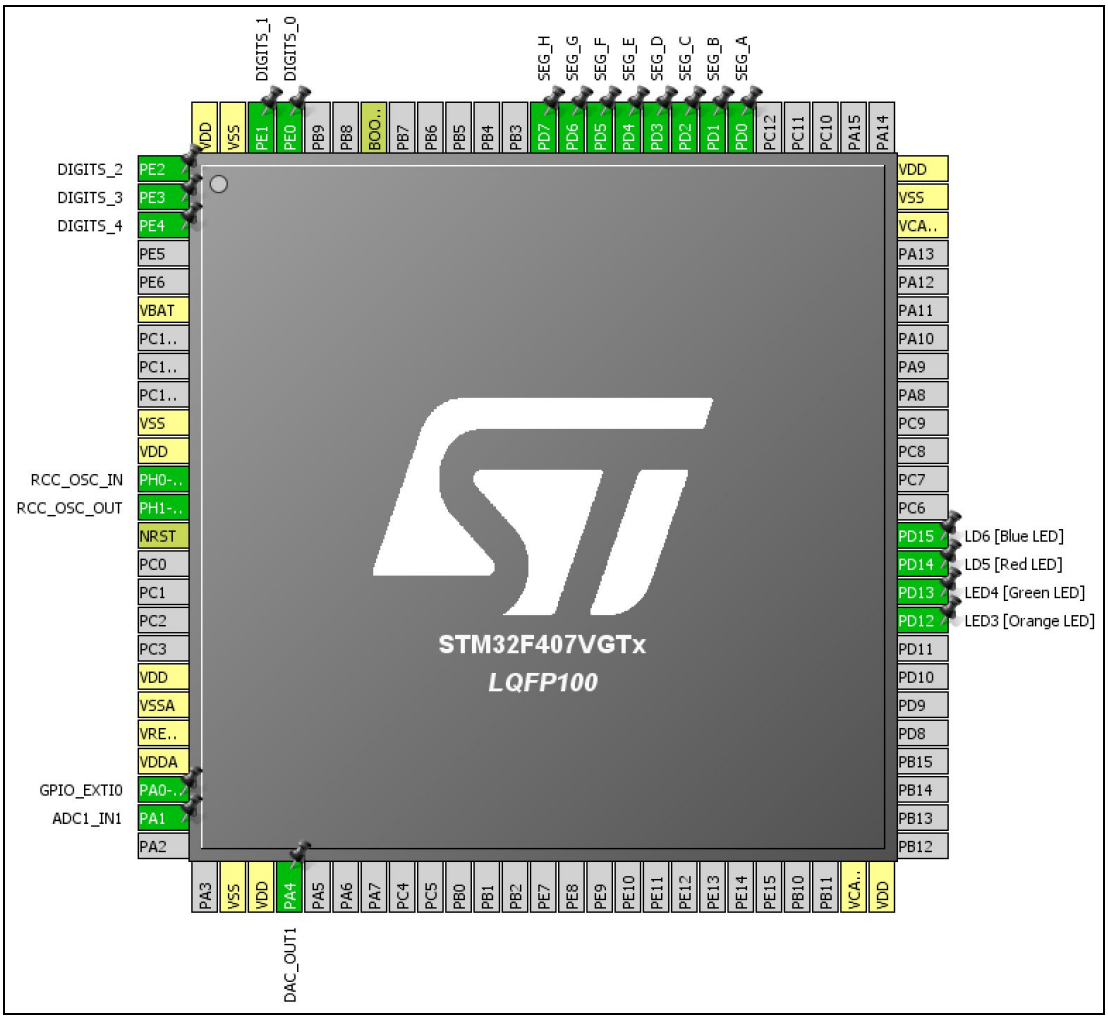

Figure 2 - CubeMX final pin layout

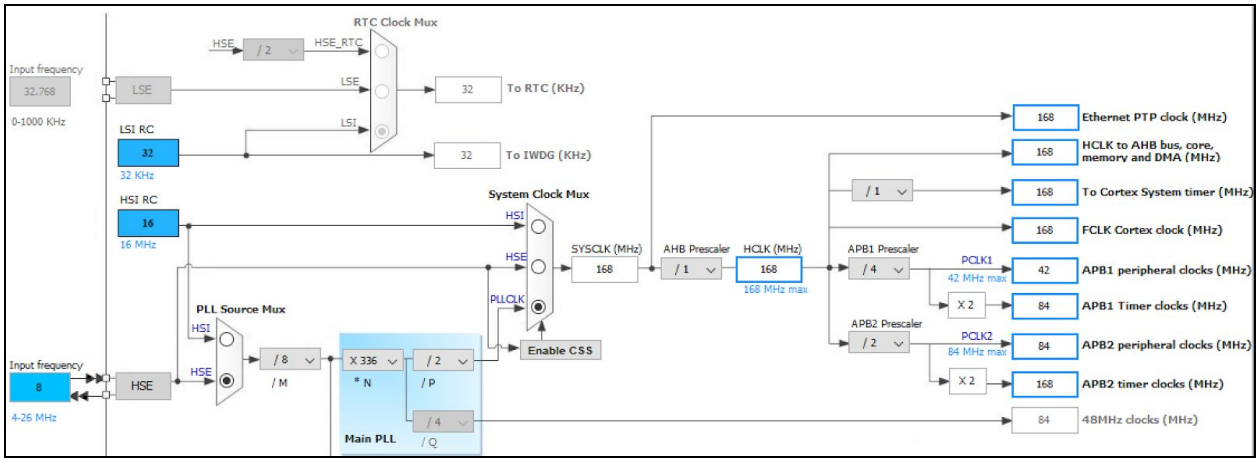

Figure 3 - CubeMX clock configuration settings

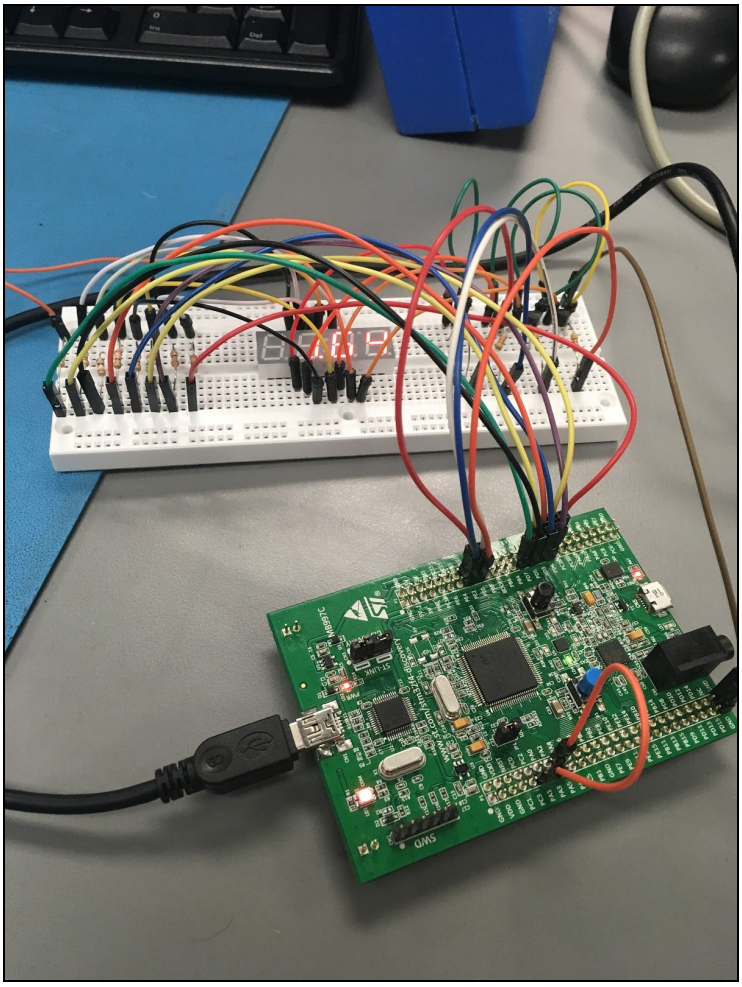

Figure 4 - Final setup of the experiment with working 7-segment display

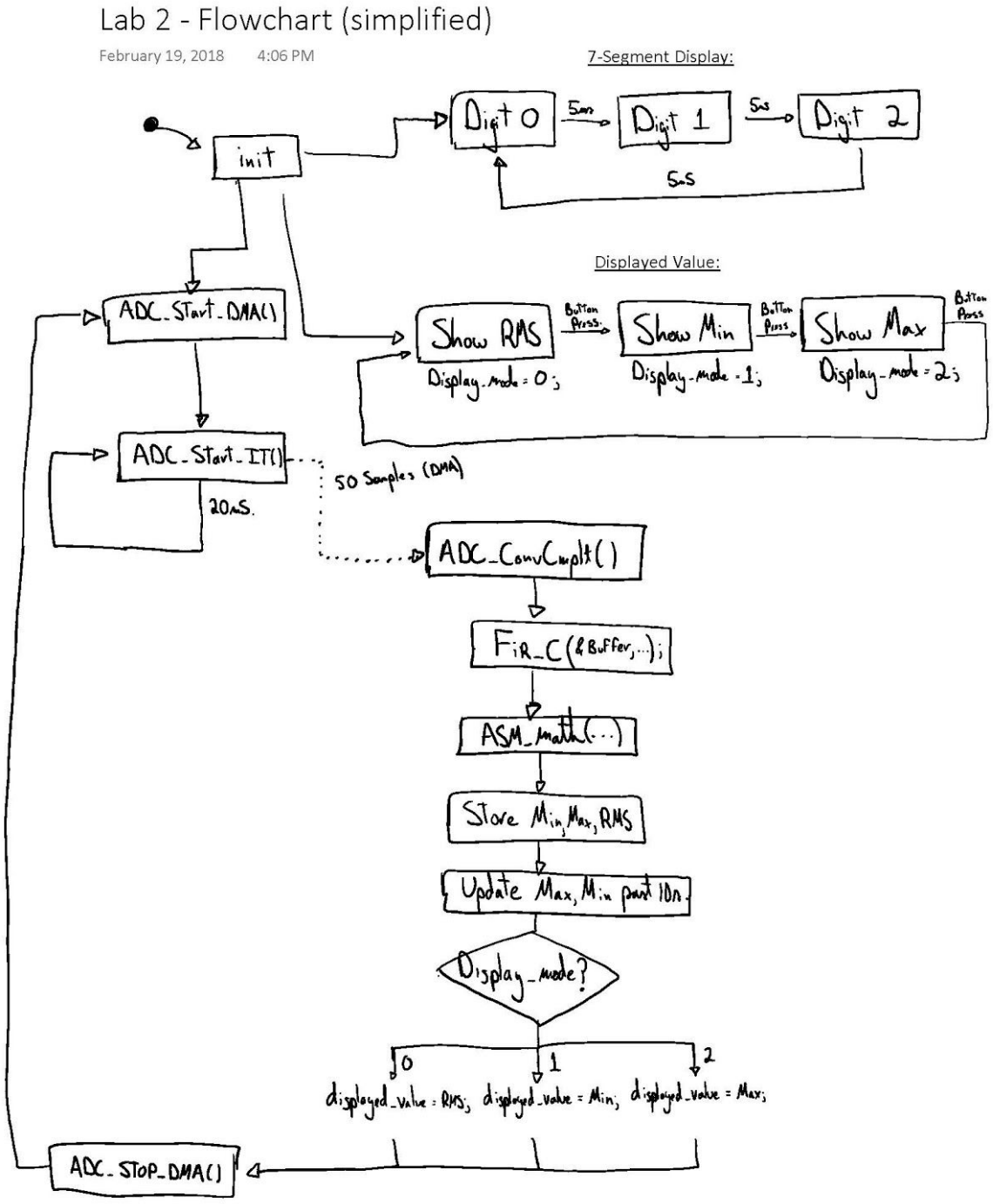

Figure 5 - Simplified flowchart